Accelerating Fast Fourier Transform with half-precision floating point hardware on GPU

Anumeena Sorna & Xiaohe Cheng
Mentor: Eduardo D'Azevedo & Kwai Wong

BACKGROUND
INFORMATION

Our project concerns a new implementation of the classical discrete Fourier Transform and the fast Fourier Transform algorithm.

Discrete Fourier Transform

Converts time domain signals to frequency domain signals according to the equation:

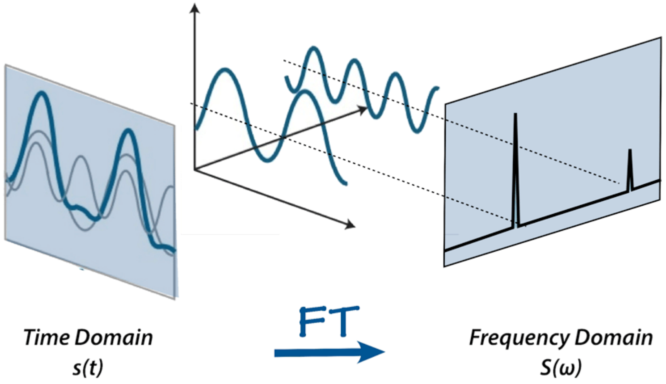
$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot e^{-j\left(\frac{2\pi}{N}\right)nk}$$

Inverse DFT:

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) \cdot e^{j\left(\frac{2\pi}{N}\right)nk}$$

Applications in:

- Convolution
- Filtrering
- Image Processing



Source: MRI Questions http://mriquestions.com/fourier-transform-ft.html

Discrete Fourier Transform

DFT can also be represented in matrix form:

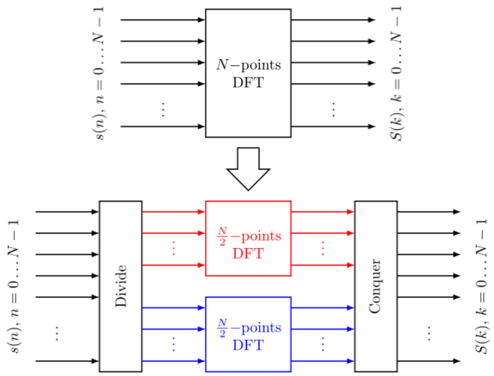
$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot W_N^{kn} \qquad W = e^{-j2\pi/N}$$

$$\begin{bmatrix} X[0] \\ X[1] \\ X[2] \\ \vdots \\ X[N-1] \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & W_N^1 & W_N^2 & \cdots & W_N^{(N-1)} \\ 1 & W_N^2 & W_N^4 & \cdots & W_N^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & W_N^{(N-1)} & W_N^{2(N-1)} & \cdots & W_N^{(N-1)^2} \end{bmatrix} \begin{bmatrix} x[0] \\ x[1] \\ x[2] \\ \vdots \\ x[N-1] \end{bmatrix}$$

Linear Transformation!

The Fast Fourier Transform

Divide and Conquer Principle



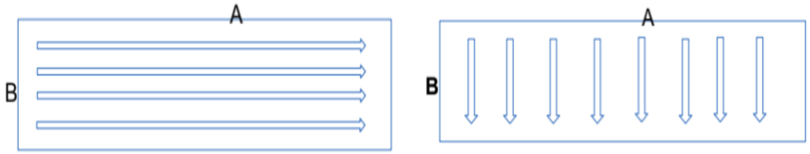
Source: DSPlib http://en.dsplib.org/content/fft_introduction/fft_introduction.html

FFT Computation requires: ~N*log(N) whereas DFT: N^2

4 Step Algorithm

Data represented as B by A matrix

1. Perform B number of A-point FFT (in parallel, stride B)
2. Perform scaling by twiddle factors exp(-(2π/N)*j*k*I)
3. Perform A number of B-point FFT (in parallel, stride 1)
4. Transpose data to form A by B matrix



Example Problem - DFT

x=[1,2,3,4,5,6,7,8]

$$W = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 & \omega^8 & \omega^{10} & \omega^{12} & \omega^{14} \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 & \omega^{12} & \omega^{15} & \omega^{18} & \omega^{21} \\ \omega^0 & \omega^4 & \omega^8 & \omega^{12} & \omega^{16} & \omega^{20} & \omega^{24} & \omega^{28} \\ \omega^0 & \omega^5 & \omega^{10} & \omega^{15} & \omega^{20} & \omega^{25} & \omega^{30} & \omega^{35} \\ \omega^0 & \omega^6 & \omega^{12} & \omega^{18} & \omega^{24} & \omega^{30} & \omega^{36} & \omega^{42} \\ \omega^0 & \omega^7 & \omega^{14} & \omega^{21} & \omega^{28} & \omega^{35} & \omega^{42} & \omega^{49} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega & -i & -i\omega & -1 & -\omega & i & i\omega \\ 1 & -i & -1 & i & 1 & -i & -1 & i \\ 1 & -i\omega & i & \omega & -1 & i\omega & -i & -\omega \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -\omega & -i & i\omega & -1 & \omega & i & -i\omega \\ 1 & i & -1 & -i & 1 & i & -1 & -i \\ 1 & i\omega & i & -\omega & -1 & -i\omega & -i & \omega \end{bmatrix}$$

where

$$\omega = e^{-\frac{2\pi i}{8}} = \frac{1}{\sqrt{2}} - \frac{i}{\sqrt{2}}$$

Matrix Multiplying x and W,

X = [36, 4 + 9.7i, -4 + 4i, -4 + 1.7i, -4, -4 – 4i, -4 – 9.7i]

Example Problem - FFT

$$X = \begin{bmatrix} 1 & 5 \\ 2 & 6 \\ 3 & 7 \\ 4 & 8 \end{bmatrix}$$

1) $X = \begin{bmatrix} \boxed{1 \quad 5} \\ \boxed{2 \quad 6} \\ \boxed{3 \quad 7} \\ \boxed{4 \quad 8} \end{bmatrix} \xrightarrow{\text{4, 2pt FFTs}} Y1 = \begin{bmatrix} 6 & -4 \\ 8 & -4 \\ 10 & -4 \\ 12 & -4 \end{bmatrix}$

2) Twiddle Factor 3)

$$W = \begin{bmatrix} W^{0*0} & W^{0*1} \\ W^{1*0} & W^{1*1} \\ W^{2*0} & W^{2*1} \\ W^{3*0} & W^{3*1} \end{bmatrix}$$

$Y2 = \begin{bmatrix} 6 & -4 \\ 8 & -2.8 + 2.8i \\ 10 & 4i \\ 12 & 2.8 + 2.8i \end{bmatrix} \xrightarrow{\text{2, 4pt FFT}} Y1 = \begin{bmatrix} 36 & -4 + 9.7i \\ -4 + 4i & -4 + 1.7i \\ -4 & -4 - 1.7i \\ -4 - 4i & -4 - 9.7i \end{bmatrix}$

RESEARCH

GOALS

- To utilize the tensor core hardware by NVIDIA
- To implement computational tricks
- To consider domain-specific requirements

Volta Architecture

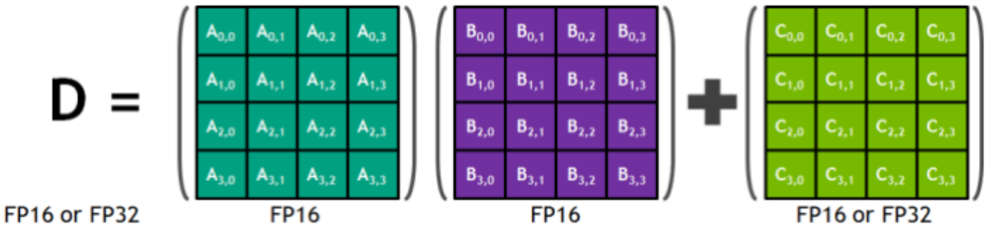
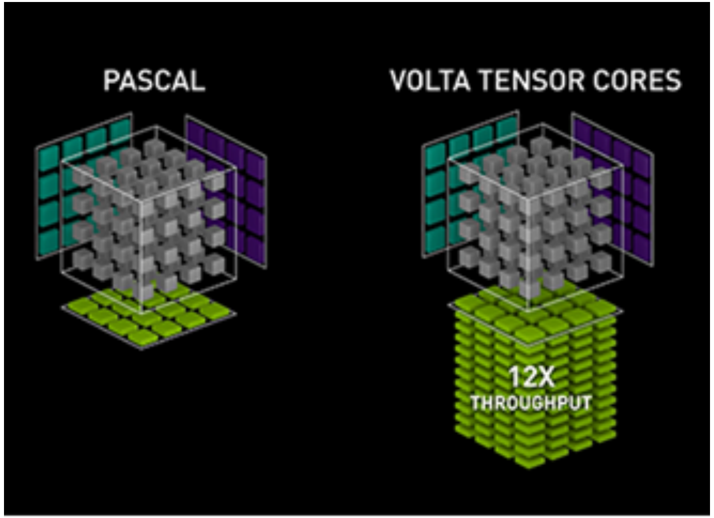


Figure 1. Tensor core 4*4*4 matrix multiply and accumulate. Source: https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/



Source: https://www.nvidia.com/en-us/data-center/tensorcore/

Tensor cores give a 8x increase in throughput using half precision input. This has been utilized by cuBLAS and cuDNN library to accelerate matrix multiplication and artificial intelligence training.

Challenge

The representation range of FP16 is roughly 6*10^(-5) to 6*10^5, which is much more limited than single precision. This degrades the precision of operations and may cause frequent overflows.

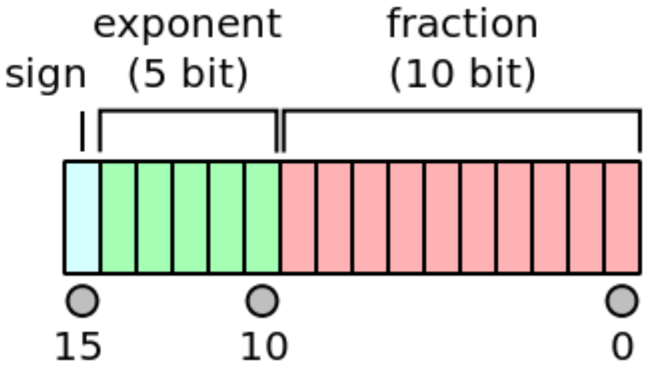


Figure 1. Half precision floating point (FP16) number representation. Source: https://en.wikipedia.org/wiki/Half-precision_floating-point_format

Single to Half Precision

To keep the accuracy, we split a FP32 number to the scaled sum of two FP16 number, and make use of the property that Fourier Transform is a linear operation:

$$x\_fp32(:) = s1\_fp32 * x1\_fp16(:) + s2\_fp32 * x2\_fp16(:)$$

and

$$\mathbf{X}\_fp32(:) = s1\_fp32 * \mathbf{X}1\_fp16(:) + s2\_fp32 * \mathbf{X}2\_fp16(:)$$

where scaling factor s1 and s2 are determined by the maximum absolute value in the original vector.

GPU Implementation

We first wrote Matlab code to test the algorithm, and will proceed to implement it with C and CUDA. We call cuBLAS library for matrix-matrix multiplication.

```
FX_re = (F4_re * X_re - F4_im * X_im);
FX_im = (F4_re * X_im + F4_im * X_re);
```

```
cublasGemmEx(handle, CUBLAS_OP_N, CUBLAS_OP_N, 4, 4, 4,
             &s1, F4_re, CUDA_R_16F, 1, X_re, CUDA_R_16F, 1,
             0, FX_re, CUDA_R_16F, 1, CUDA_R_16F, CUBLAS_GEMM_DEFAULT)
```

Further acceleration

3M algorithm, 2D fft & in-place transformation, partial FFTs

3M Algorithm

$$z = (a + ib)(c + id) = ac - bd + i(ad + bc)$$

$$z = ac - bd + i\left[(a + b)(c + d) - ac - bd\right]$$

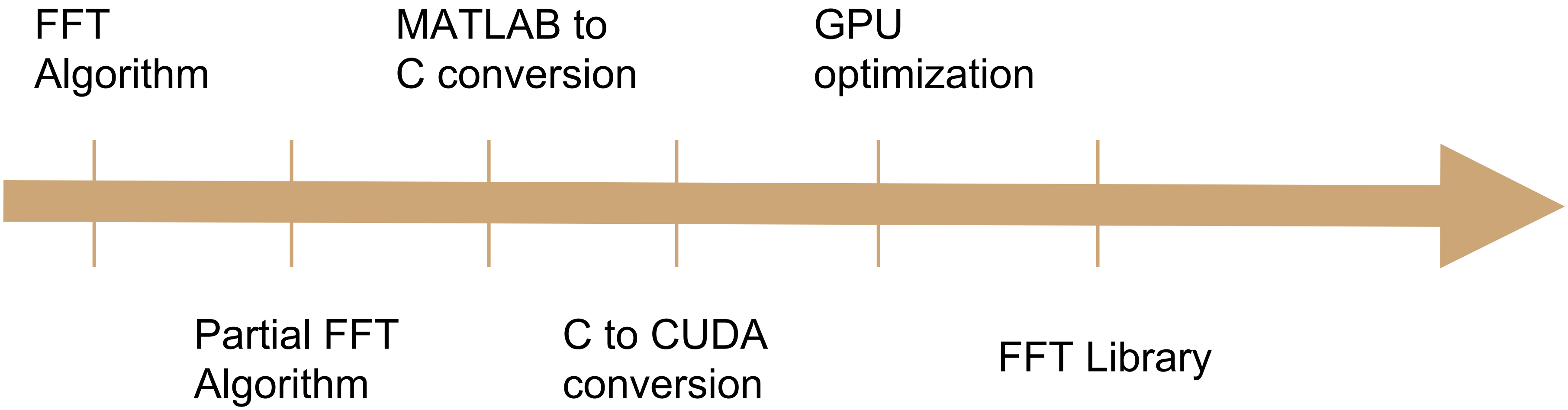
$$\begin{aligned} T_1 &= A_1 B_1, \qquad T_2 = A_2 B_2, \\ C_1 &= T_1 - T_2, \\ C_2 &= (A_1 + A_2)(B_1 + B_2) - T_1 - T_2, \end{aligned}$$

Partial FFTs

In some applications we only require a portion of the matrix with FFT results

An algorithm can be used to efficiently compute only required portions instead of usual method of computing all and discarding unnecessary FFT values

Current Progress & Future Work



Q & A

Any questions?